CS10720 Problems and Solutions

Thomas Jansen

Today: Arrays

Remember Mid-Semester Survey http://goo.gl/forms/x2lvCT8ueU

February 29th

Reminder: In-Class Test

Remember in-class test today in a week (7th March 2016)
here, at the usual time (5.10pm–6pm)

- Please, be here on time! (Preferably by 5.05pm.)
- Please, bring something to write: blue or black pen.
  (I'll provide the paper.)
- Relevant material: everything covered in lectures and practicals between 28.01.2016 and 24.02.2016.
  This means not matrices, not arrays.
- Use example problems on Blackboard for your preparation.
- Please, take this seriously.
  It contributes 30% to your module mark.

Plans for Today

Arrays in C

Remember `int v[6];`
creates array of size 6, capable of holding 6 `int`s
one each at `v[0]`, `v[1]`, `v[2]`, `v[3]`, `v[4]`, `v[5]`

What does this mean internally?

	0x200000	0x200004	0x200008	0x20000C	0x200010	0x200014
v:	3	4	5	42	7	8
	0	1	2	3	4	5

`a = 0x200000`

Assume `sizeof(int)`=4

Consider

```
int i;
int *a;
for ( i=0; i<6; i++ )
  v[i]=i+3;
a = (int *)v;
a[3]=42;
```

Summary One-Dimensional Arrays

- `type a[size];`
  array `a` of type `type` and size `size` occupies
  `sizeof(type)` · `size` consecutive bytes in memory
- `a[8]` gives access to 9^{th} item in `a`
  Careful: no bounds checks!
- `a` gives starting address
- `&a[3]` gives address of 4^{th} item in `a`
  Careful: no bounds checks!
- `type *b = (type *)a;`
  `b` gives starting address of array `a`
- Pointer arithmetic: `b+3` gives address of 4^{th} item in `a`
  Careful: no bounds checks!
- `*(b+3)` gives access to 4^{th} item in `a`
  Careful: no bounds checks!
- Consequence `*(b+3)=8;` and `b[3]=8;` and `a[3]=8;`
  all equivalent

Two-Dimensional Arrays in C

Fact `int m[4][8];`

creates array of size 4×8, capable of holding $4 \cdot 8 = 32$ `int`s one each at `m[0][0]`, `m[0][1]`, `m[0][2]`, ..., `m[0][7]`,
`m[1][0]`, `m[1][1]`, `m[1][2]`, ..., `m[1][7]`,
`m[2][0]`, `m[2][1]`, `m[2][2]`, ..., `m[2][7]`,
`m[3][0]`, `m[3][1]`, `m[3][2]`, ..., `m[3][7]`

What does this mean internally?

```
int m[4][8];
int i, j;
int *a;
for ( i=0; i<4; i++ )
  for ( j=0; j<8; j++ )
    m[i][j] = 10*i+j;
a = (int *)m;
for ( i=0; i<4*8; i++ )
  printf("%d ", a[i]);
```

Output

0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 20 21 22 23 24 25 26 27 30 31 32 33 34 35 36 37

Observations

- two-dimensional array is continuous block in memory
- address of `m[i][j]` is $m + \mathrm{num}_j \cdot i + j$ with $\mathrm{num}_j = 8$ in this case

Applying Two-Dimensional Arrays

Remember matrices

Observation matrices and two-dimensional arrays
'match made in heaven'

```
#define n 10
#define m 20
#define o 30
double s, a[n][m], b[n][m], c[n][m], d[m][o], e[n][o];
int i, j, k;
/* matrix addition:  C = A + B */
for ( i=0; i<n; i++ )
  for ( j=0; j<m; j++ )
    c[i][j] = a[i][j] + b[i][j];
/* scalar multiplication:  C = s · A */
for ( i=0; i<n; i++ )
  for ( j=0; j<m; j++ )
    c[i][j] = s * a[i][j];
```

Applying Two-Dimensional Arrays (continuted)

Remember matrices

Observation matrices and two-dimensional arrays
'match made in heaven'

```
#define n 10
#define m 20
#define o 30
double s, a[n][m], b[n][m], c[n][m], d[m][o], e[n][o];
int i, j, k;
/* matrix multiplication:  E = A · D */
for ( i=0; i<n; i++ )
  for ( j=0; j<o; j++ ) {
    e[i][j] = 0; /* initialise sum as 0 */
    for ( k=0; k<m; k++ )
      e[i][j] = e[i][j] + a[i][k] * d[k][j];
  }
```

Multi-Dimensional Arrays

Fact there is no reason to stop at 2 dimensions

Fact works with arbitrary number of dimensions
e. g., `double a[50][40][30][100];`
creates 4-dimensional array `a`
with room for $50 \cdot 40 \cdot 30 \cdot 100 = 6\,000\,000$ `doubles`
accessible as you would imagine, e. g., `a[23][38][15][98]`
at address $a + 100 \cdot 30 \cdot 40 \cdot 23 + 100 \cdot 30 \cdot 38 + 100 \cdot 15 + 98$

Why would anyone need 4-dimensional arrays?

Remember image processing
image can be encoded as 2-dimensional array
sequence of images (e. g., animated GIF or movie)
can be encoded as 3-dimensional array
sequence of 3-dimensional images (e. g., 3D movie)
can be encoded as 4-dimensional array

Limits of Static Arrays

Observation `double a[50][40][30][100];`
requires the values 50, 40, 30, 100 to be known at compile time

What if I do not know the required size of my array because that only becomes known at run time?

Two Options

1. use safe upper bounds instead
2. leave size undetermined at compile time and create space when program is running

Problems with

1. perhaps no safe upper bounds known
   wasteful if upper bounds are much larger than actual values
2. 'I don't know how to do that.'

Let's solve problem 2!

One-Dimensional Dynamic Arrays

Problem Create something like `int a[n];`
where the value of `int n` is determined at run time

Remark not a problem because C supports this directly
since 1999 (known as C99)
your compiler just needs to support C standard
less than 25 years old
(because the standard before was C89 or C90)
Problem Visual Studio 2013 does not

Can we not use a modern C compiler?

We could but

1. we want to be able to write portable code that runs on as many machines/compilers/operating systems as possible
2. what we get to know now is universally useful

One-Dimensional Dynamic Arrays

Problem Create something like `double a[n];`
where the value of `int n` is determined at run time

Solution `void *malloc(size_t size)`
for allocating the space you need and
`void free(void *ptr)`
for releasing allocated space (i. e., giving it back to OS)

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */
int main(void) {
  double *a;
  int    n;
  printf("Enter number between 1 and 100:  ");
  assert( scanf("%d", &n) == 1 ); /* reads 1 number; stops program if this fails */
  assert( (n>0) && (n<101) ); /* stops program if n not between 1 and 100 */
  a = (double *)malloc(sizeof(double)*n); /* allocates space for a */
  assert( a!=NULL ); /* stops program if space was not available */
  a[n-1] = 42.12; /* stupid example */
  free(a); /* frees space used for a again */
  return 0; /* exits program without error */
}
```

Summary & Take Home Message

Things to remember

- static arrays: one- and multi-dimensional
- implementing matrix operations with two-dimensional arrays
- dynamic one-dimensional arrays

Take Home Message

- Arrays are incredibly useful data structures.
- Dynamic arrays allow for responsible use of space.
- Index computations are not difficult
  and help writing efficient programs.

Lecture feedback `http://onlineted.com`